Chopsticks AI: Solving a Zero-Sum Game Using Decisions Trees and Minimax Algorithm

Bryan S Hutto

University of North Georgia

Department of Computer Science and Information Systems

82 College Circle, Dahlonega, GA 30597

bshutt6946@ung.edu

ABSTRACT

This paper describes an intelligent program designed to play and win a variant of the traditional hand game “Chopsticks.” The program runs the game, managing the player input in accordance with the game’s rules as well as determining and effecting the best move for the computer player. It populates a fully descriptive decision tree of every possible state from a given starting state and to a given depth, where depth is equal to number of turns from the starting state. A fairly standard minimax implementation then selects the best move from the current state using the decision tree created from that state.

Categories and Subject Descriptors

I.2.1 [Artificial Intelligence]: Applications and Expert Systems – *games*

G.1.2 [Mathmatics of Computing]: Approximation – *minimax approximation and algorithms*

K.8.0 [Computing Milieux]: Personal Computing – *Games*

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

General Terms

Algorithms, Games, Decision Tree

Keywords

Chopsticks, game, decision, minimax, algorithm, solve, optimal play, input handling.

1. Introduction

This paper will explain an effective but improvable implementation of a recursive minimax algorithm applied to decision tree. The subject of this program is a modified version of Chopsticks. The original version of Chopsticks has been solved; the seconded player will always win given optimal play, as demonstrated on wikiHow.com [1]. But can the same be said for this particular variation? If so is a simple minimax algorithm enough to achieve it? This project attempts to answer this question by way of induction, building an ambitious implementation and testing it against human players and, in the future, itself.

2. RELATED WORK

Very little outside material was utilized in the construction of the program barring one very large exception, aside from general knowledge Python information obtained from the documentation and various online help forums. Much credit must be given to Trevor Payne et al. [2], who’s tutorial gave this researcher a better general understanding of what minimax implementations might look like in practice, as well specific ideas such as using python’s list function to stored and add children to a node and multiplying the value of a state by a negative or positive one depending on who’s turn the node is a child of in the minimax algorithm.

3. CHOPSTICKS

The rules for our splitting variation of Chopsticks are as follows:

Both players start with 1 finger extended on each hand.

Players alternate turns.

During a turn a player may:

-Tap the opponent with one of their hands, at which point the opponent’s hand becomes the sum of the "attacking and "defending" hands, while the attacker's hand remains the same.

-If the player has an even number of extended fingers on one hand and none on the other, they may "split" the number across both hands. So 0-4 becomes 2-2 and 2-0 becomes 1-1

If the defender has exactly 5 fingers extended after an attack all fingers on that hand are retracted and it becomes a "dead" hand (a dead hand can only be revived by means of a split).

If the defender’s count goes over 5 after an attack, his count now becomes the sum of the attacking and defending hands minus 5, so 4 attacking 4 becomes 3, 3 attacking 4 becomes 2, etc...

The first player to have 2 dead hands loses.

In this program the player indicates his move via entering a string: “ll”, “lr”, “rr”, “rl”, “spl” or “pass”, where “ll” is left attacks left, “lr” is left attacks right and so on. A split is indicated by “spl”, and “pass” forfeits the turn.

4. APPROACH

4.1 The Decision Tree

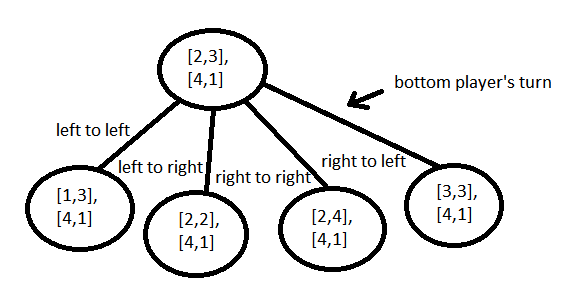


Figure 1: An example of a node and its children in the tree

The tree is constructed procedurally by means of a method “createChildren(),” which automatically executes when a root node is created and again for every child created in the method to a given depth. The current implementation of creatChildren() is rather monolithic. An if statement determines who’s turn it is, routes the program to the appropriate subroutines, then generates each state achievable from the current node in an arbitrary ordering. For each of these states the current node’s “children[]” attribute has a new node with the newly calculated state appended to it. Since the creation of a new “Node” object automatically calls createChildren() on itself, this method works recursively to populate the tree in a depth-first manner, switching the player turn and therefore the set of states available at each depth. Only after the selected depth is reached does the process return to the next subroutine in the parent node’s creatChildren(), and so on.

4.1 Depth Selection

From our textbook [3], we know the time complexity for a depth-first search, which our minimax algorithm uses, is O(bm), where b is the branching factor and m is the depth. From this we see that the time needed to calculate the appropriate move will increase exponentially with depth. Thus, selecting a depth which both significantly outpaces the human player’s ability to look ahead into the game states while still maintaining a reasonable run time is of paramount importance. The current version of the program builds a tree to a depth of 9 for each node examined and runs a minimax to depth of 8 on the tree for each child of said node, however we have discovered no marked increase in utility in this form from a 7/6 depth scheme and would recommended this later setting for researchers with less than optimum processing power.

4.2 The Minimax Algorithm

The miniMax() function takes a node generated from the current state, and therefore the entire decision tree from that state, as well as a depth parameter and a turn parameter. Every time the method is called it creates a variable “goal” and sets it equal to the best value for the opposing player, based on the current value of “turn.” The method then calls itself on each child of the current node, passing a decremented value for depth and an alternated value for turn, thereby traversing the tree structure in the same depth-first manner seen in createChildren(). The recursion ends and returns the value of the current node (0, -1000, or 1000) when a node is passed with a depth of zero or a “strength” attribute value of +/- 1000, the later indicated a win state for, positive, the computer or, negative, the player. When this happens the value is received by the previous instance, multiplied by the player turn at the current depth of recursion and is compared against the current value of goal, replacing it if it is greater or lessor, depending on the value of turn. This will continue until a final minimax value is returned for the originally passed state. In aiMove() this minimax value is compared against the minimax values for the remaining children of the current game state.

4.3 Optimizations

In the recent live demo of the Chopsticks Ai some serious issues with the current design were brought to light. The most egregious of which being the programs inability to distinguish between children with a minimax value of zero that continued to advance the game state and children which would result in a loop given optimal play by the human. A lessor criticism is the inability of the Chopsticks Ai to distinguish between a move that will result in a win in one or two moves and a one that will result in a win at the maximum depth of the tree. Both of these flaws have been addressed in the most recent update.

*4.3.1 Randomized Child Selection*

In this version the children examined in aiMove() are stored in a list, “bestChoice[],” if they meet the requirement of having a minimax value greater than or equal to the “bestStrength” variable (defaulted to zero and updated when a child meets said requirement). The program then randomly selects one of the qualified members of the list as its move, rather than simply selecting the most recent best choice, as previous versions had done. In this schema, while a sufficiently cognizant human player could theoretically prolong the game, it will be exponentially more difficult to do so.

*4.3.1 The SuperChoice Override*

In order to encourage the program to ‘go for the throat’ if a win is possible in the next few moves, an override has been implemented. Now aiMove() also runs an additional minimax method on each child while passing a depth of 3. If this value comes back as 1000, indicating a win in the next four moves, the SuperChoice selector is engaged and that child is instead chosen as the computer’s next move.

5. RESULTS

As with previous versions, the current Chopsticks Ai has not been demonstrated to be beatable by a human player. Additionally, it has not been demonstrated to be possible for a human player to infinitely prolong the game by finding loops. Included with this paper will be both the finished product and a testing version which outputs the minimax values being compared, the qualifying nodes, the node selected, and the use of the SuperChoice override. From this data one can see that, unlike other variations of the Chopsticks hand game, the bias appears to be in favor of the first player. If the human player goes first the computer will be choosing among both -1000 and 0 strengths, while if the player chooses to pass on their first turn the computer will be choosing only from nodes with 0 strengths, as you would expect for the first move of the game. However, much more testing is required to establish this definitively.

6. CONCLUSIONS

This program can offer an excellent introduction to the use of a minimax algorithm for evaluating a decision tree. As such, it would make a useful reference for students trying to implement something similar or just trying to get a better grasp on how these concepts work in practice. It is also extensible and could be used as a foundation for experimenting with different decision making heuristics.

7. FUTURE WORK

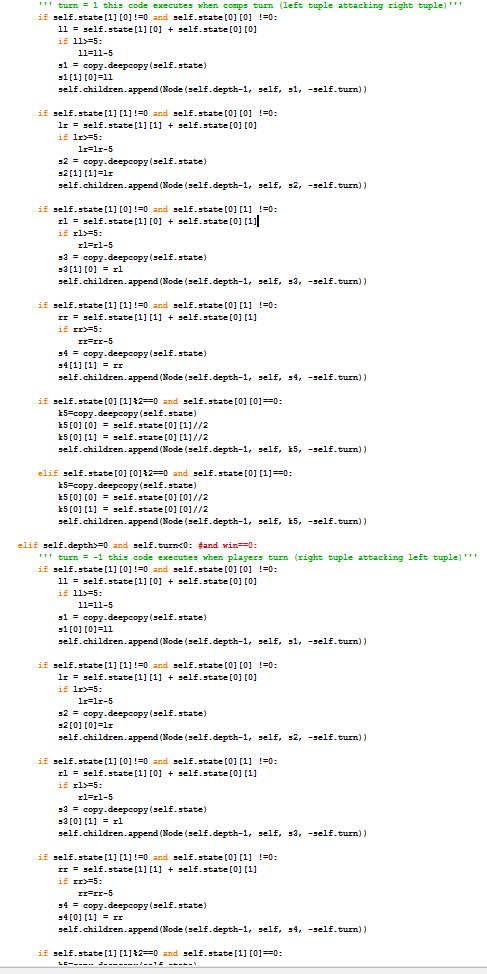


Figure 2: Part of the monolithic tree builder

Significant credit must be given to Nathun Tippie [4], for assisting this researcher in python code optimization. His influence can be observed in the method make\_move(), which handles the players turn. Previously this function had much the same structure as createChildren(), but now it utilizes Python dicts to associate a particular letter in the left/right attack indicator string with the corresponding index in the state object, then it checks the validity of the move and generates the result according to this relationship. This improves extensibility because now, for example, the function needs only one value changed to work equally well with a different number of fingers per hand. In the future, the make\_move() function is intended to handle the tree generation as well as the player turn, utilizing list reversal to handle alternating turns, thereby significantly reducing the lines of code needed and increasing extensibility. However, due to time constraints, the tree building in creatChildren() is still hard coded for this particular version of Chopsticks.

An important experiment this project did not have time for was rewriting the program so that it can play a game against itself. The result of such a game would tell us quite a bit about this variation of Chopsticks’s solvability and would presumably answer the question of which player actually has the advantage. Optimal play is hard to simulate as a human tester, and a computer verses computer situation should reveal if a win can be forced or if the game will always be prolonged indefinitely.

This program also represents an ideal platform for experimenting with any number of math games, and the intended, more extensible version could serve as a testing ground for game creators, as it will include a more user-friendly interface and the option for human verses human play.

7. REFERENCES

[1] WikiHow: How to Always Win Chopsticks, n.d. Retrieved March 5th, 2016, from <http://www.wikihow.com/Always-Win-Chopsticks>

[2] Payne, Trevor. Let’s Learn Python #21 – Min Max Algorithm, Febuary 1st, 2014. Retrieved March 5th, 2016, from <https://www.youtube.com/watch?v=fInYh90YMJU>

[3] Stuart J. Russell and Peter Norvig. 2003. *Artificial Intelligence: A Modern Approach* (2 ed.). Pearson Education. pp. 77

[4] Tippie, Nathun. University of Georgia B.S. ntippie@gmail.com